**Artificial Intelligence** **(Harvard CS182, Fall 2015)**

# Lectures Notes on MDPs

*Alexander Rush*

## Contents



## 1 Introduction

Throughout the first half of the class we have made the assumption that our actions were deterministic. When we decided to take a turn in a path-finding problem, we followed that new path. When we filled in a variable in Sudoku, that variable was assigned a new value.

For many problems this is the right abstraction to be using, e.g. the Sudoku variable does reliably get filled in with the value we decide. However, for many other problems a decision does not determine the outcome. We hope that taking one path may lead us to the goal in 10 minutes,

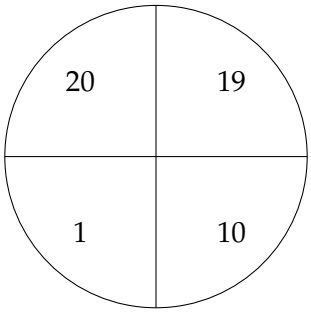
or maybe there is traffic and it will take 15, or we will be forced to take a detour and end up in a nearby location.

The second half of the class will focus primarily on this crucial aspect of AI which we will refer to as *uncertainty*. The main extension we will make is to incorporate non-deterministic outcomes and noisy sensor information into the core models we have explored to this point. The tool we will use to specify the type of uncertainty will be probability distributions. In this set of lectures, we will consider adding uncertainty into classical search which will yield a model called Markov decision processes. We will also see a very different algorithm used to solve these models known as value iteration. In the next set of lectures we will consider uncertainty in aspects of the surrounding world.

## 2 Rewards Under Uncertainty

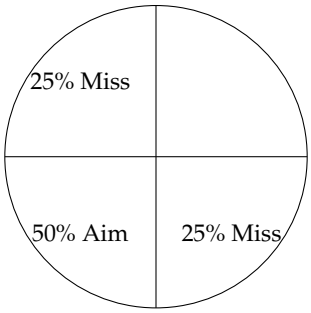
Before diving into to the main focus of this lecture, let us consider a simple example of rationality under uncertainty. By uncertainty we mean worlds with non-deterministic outcomes, and by rationality we mean behaving to maximize our utility, which in this section we will call reward (since there is no opponent). In fact, this example will be very reminiscent of the ExpectiMax formalism for games of with randomness, and much of this section takes off from where we left off with Expectimax. However for simplicity we will focus on uncertainty and assume there is no adversary.

Throughout this lecture, we will consider Darts as an example of a one-player game against nature. For this section, consider a simple one-shot game of darts. Here our goal is to score the highest number of points from a board. We will call the value we hit, our reward (similar to utility in game-playing). Here is the dart board:




As a deterministic game, darts is frankly terribly boring. The strategy is obvious, we just pick the highest scoring cell. However, in practice I am not very good at darts. Even if I aim for a square (say 1) there is a rather high chance that I will miss. In fact let's assume that the chance of missing to the clockwise and counter-clockwise looks like the following:[1]

[1] I am being kind here, there is certainly a non-zero chance that I miss the board entirely.




That is if we aim for the bottom-left, we will hit that square with probability 50% and top-left with probability 25% and bottom-right with probability 25%. Using this information we can calculate the expected (local) reward for each of the possible aims we might takes.

**Definition 1** *The expected local* reward *is the expectation of the reward gained from taking an action.*

$$\textsc{ExpectedReward}(a) = \mathbb{E}_{o|a}(R(o)) = \sum_o p(o|a)R(o)$$

where $o$ is the outcome and $p(o|a)$ is the conditional probability of each outcome given the action that we took (aiming for a square). (For now think of $p(o|a)$ as being a function given to us, we will define conditional distributions more rigorously in the next lecture.) The *optimal decision* to take under this model is the one that maximizes expected reward.

$$a^* = \underset{a \in \textsc{Actions}}{\arg\max}\ \textsc{ExpectedReward}(a)$$

For the example we can simply calculate by hand the value for all the possible actions (aiming positions). This gives expected rewards of:

$$\begin{aligned}
\textsc{ExpectedReward}(\textsc{Bottom-Left}) &= 0.5 \times 1 + 0.25 \times 20 + 0.25 \times 10 = 8 \\
\textsc{ExpectedReward}(\textsc{Top-Right}) &= 0.5 \times 19 + 0.25 \times 20 + 0.25 \times 10 = 17 \\
\textsc{ExpectedReward}(\textsc{Top-Left}) &= 0.5 \times 20 + 0.25 \times 19 + 0.25 \times 1 = 15 \\
\textsc{ExpectedReward}(\textsc{Bottom-Right}) &= 0.5 \times 10 + 0.25 \times 10 + 0.25 \times 1 = 7.75
\end{aligned}$$

So under this reward function and the uncertainty associated with the problem the optimal action is to aim for position TOP-RIGHT (19) with expected reward of 17 points. The equivalent ExpectiMax interpretation is shown in Figure 1.

## 3 Sequential Decision Problems

The focus of this chapter will be extending rationality under uncertainty to sequential problems. For one-shot uncertainty we can often use brute-force to calculate expected rewards. But the case becomes more complex when we introduce uncertainty into the classical search formalism. Here we have to consider how uncertainty compounds over many decisions.

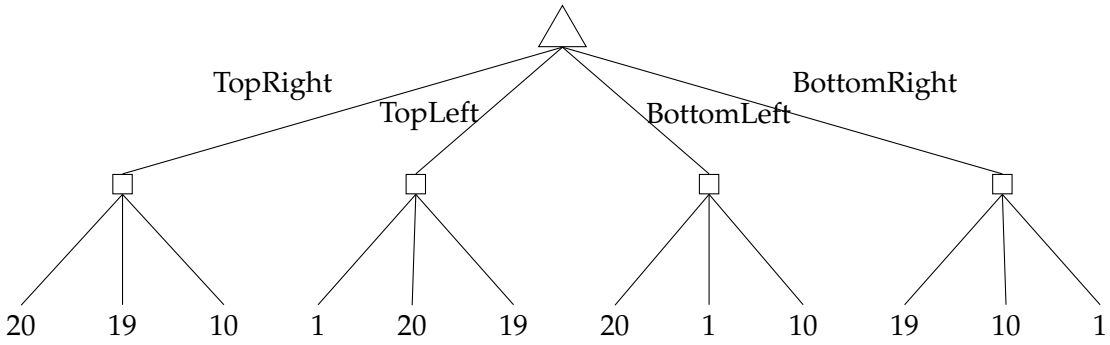



*Figure 1: Expectimax interpretation of one-shot darts.*

### 3.1 Running Example: Darts

Our running example will continue to be darts. However let's play an actual game of darts. Here are the rules:

- We start with a target value of 301.
- Each time you throw a dart, your score drops by the number of points in the outcome.
- Unless it would go below 0, then it stays the same.
- When you hit exactly 0, the game is over.
- The goal is to use as few throws as possible.

The ending condition of Darts makes the game much more interesting. The action that you take at any point in time depends on your current score and the chance of hitting the right target, but also on anticipation of future outcomes. Because of this we refer to the problem of optimal play as sequential decision making.

### 3.2 Markov Decision Process

Let's begin by defining the formalism used for sequential decision making, known as a Markov decision process or MDP. An MDP is a search problem with an uncertain transition model. That is, given a state and an action, there is a probability distribution over possible next states (as opposed to search, when the next state was determined).

| Name | Type | Description |
|---|---|---|
| State space | $\mathcal{S}$ | The set of all possible states of the world. |
| Action space | $\mathcal{A}$ | The set of all possible actions of the world. |
| Action model | $\text{ACTIONS} : \mathcal{S} \mapsto 2^{\mathcal{A}}$ | Specifies which actions are applicable at a given state. |
| Transition model | $p(s'\|s, a)$ | Specifies the probability of ending up in state $s'$ when in state $s$ and taking action $a$. |
| Initial state | $s_0 \in \mathcal{S}$ | The starting state of the world. |
| Goal state | $\text{GOAL} : \mathcal{S} \to \{0, 1\}$ | Predicate indicating if a state is terminal. |

Since steps that are no longer deterministic based on the actions, we need to be more careful when specifying our cost and utilities.

| Name | Type | Description |
|---|---|---|
| Reward Function | $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ | The reward $R(s, a, s')$ of taking action $a$ in state $s$ and reaching state $s'$ in the problem. |

Unlike search we will be maximizing reward (as in the above example). Also the reward function takes into consideration the previous and next states as well as the action.

Now we give a couple definitions. These roughly generalize the core concepts of search to the MDP environment.

**Definition 2** *A* history/trajectory *is a sequence of states and action, $(s_0, a_0), (s_1, a_1), \ldots$, visited up until a current point (equivalent to a path). (However note unlike a path in search we do* not *get to preselect the history, different histories can come from the same set of action choices.).*

**Definition 3** *We say a probabilistic model is* Markovian *if the future is independent of the past given the present. Informally if our history is $(s_0, a_0), (s_1, a_1), \ldots, (s_i, a_i)$ then the transition model $p(s'|s_i, a_i)$ does not depend on any of the other states or actions before $s_i$. (We will define this rigorously in the next lecture.)*

So for MDP:

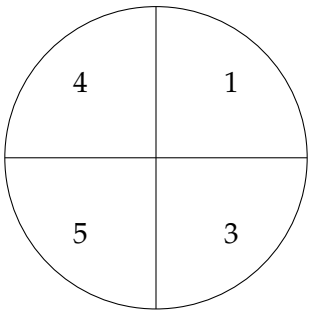
- Markov refers to the distribution used for the transition model.
- decision refers to maximizing reward under uncertainty.
- process refers to the fact that it is a sequential model as opposed to one-off.

### 3.3 Example

Let's put everything together and consider our dart's example.

| Name | Type | Description |
|---|---|---|
| State space | $\mathcal{S}$ | Current score. |
| Action space | $\mathcal{A}$ | Positions to aim for. |
| Transition model | $Pr(s'\|s,a)$ | Prob of *getting to* score $s'$ when at score $s$ and aiming for $a$. |
| Initial state | $s_0 \in \mathcal{S}$ | The starting score (301). |
| Goal state | $\textsc{Goal} : \mathcal{S} \to \{0,1\}$ | A score of 0. |
| Reward Function | $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ | Always -1 (more turns is worse) |

Now let's consider an explicit example. For simplicity we will start with a target score of $s_0 = 10$, our dart board will have the following values, and the transition model will be the same as above:




At state $s_0$ with score 10, the goal will clearly be to aim for position BOTTOM-LEFT on the board. With prob 50% this will take us to score 5, with prob 25% to score 6, and with prob 25% to score 7. Each of these will lead to a reward of -1.

But now consider a state with score 3. If we aim for position 3 on the board, with prob 50% this will take us to score 0 and a goal state, with prob 25% we will hit 5 and remain at score 3, and with prob 25% we will move to score 2 (which is actually a tough state to be in). All of these will also lead to a local reward of -1.

What is important to note, is that unlike the initial example, we cannot simply select the action to maximizes local expected reward. Because this is a sequential process, we need to take into consideration the expectation of the future reward at the state the we end up in. *Is it worth going for the goal this turn if there is some chance of ending up in a position that would be difficult to get out of?*.

An optimal policy maximizes the expected total reward as opposed to the being greedy at each position.

# 4 Decision-Making in MDPs

While an MDP looks superficially similar to search, things diverge significantly when we start considering solving an MDP. For many search problems we were able to find the optimal solution without even examining much of the search space (states and actions). The solution returned only consisted of a single path. However for MDPs it is quite possible that due to the uncertainty of the model, our trajectory could go far afield. Our solution must deal with making decisions under all possible occurrences. We will therefore need a different abstraction for a solution.

## 4.1 Policies

Instead of working with paths/histories at all, we instead focus on developing policies:

**Definition 4** *A* policy *is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$ specifying the action decision to be made at every state of the MDP.*

Of course you can develop policies for classical search as well (we roughly do this for game-playing without depth-limit or alpha-beta pruning), but since we assume actions are deterministic, it is not necessary to consider states we will not visit.

Given a policy and an MDP we also have a distribution over all trajectories, formed by combining local distributions with the action selected by the policy at each state.

We can also sample trajectories $(s_0, a_0), (s_1, a_1), \ldots \sim MDP(\pi)$ from this distribution by sampling the next state based on the current state and the action given by the policy $s_{i+1} \sim p(s'|s_i, \pi(s_i))$.

Define the expected reward for a policy as a weighted expectation of the reward under this distribution:

$$U^{\pi}(s_0) = \mathbb{E}_{(s_0,a_0),(s_1,a_1),\ldots\sim MDP(\pi)}(\sum_{i=0}^{\infty} \gamma(t) R(s_i, a_i, s_{i+1}))$$

Where the expectation here is over all possible trajectories and $\gamma$ is a function of the timestep (discussed in the next section).

The main algorithmic aim of MDPs will be to develop the optimal policy based on this distribution:

**Definition 5** *The* optimal policy *is the policy such that the expectation of the reward of histories taken from the policy is maximized.*

$$\pi^* = \arg\max_{\pi} U^{\pi}(s_0)$$

Of course we have yet to discuss how to compute this function $U$ which is the core difficulty of MDPs.

### 4.2 Horizons and Discounting

Unlike search where paths were finite, we are now allowing the possibility of infinite histories in the reward function, (although we are assuming policies are finite). The use of infinite histories will complicate our algorithms, so we now discuss two ways to handle them:

- Finite Horizon: Limit histories to a finite length. Depending on the current time step we only look from the current state to the end of the finite length cutoff. This removes any infinitely long history from any of the calculations. Done and done.
- Infinite Horizon: We allow infinite histories and take into account an infinite series of future rewards. Now have to deal with the case of a positive-reward loop which could allow infinite reward.

Naturally the first case seems easier; however it introduces a nasty issue. Let's say we are at state $s$ at time-step $i$, and we want to know our expected future reward. If our timestep is near the horizon, then we will want to act aggressively to reach a goal state sooner, but if we are far away we may try to accumulate more reward before reaching a goal. Unfortunately this means the expected reward now depends on the current timestep ! We say that policies under this objective are *non-stationary*.

For this reason counter-intuitively, if we can deal with the infinite reward issues, the infinite horizon is actually *easier* to reason about . To do this, we redefine the reward of a history to utilize *discounting*.

$$\sum_{i=0}^{\infty} \gamma^i R(s_i, a_i, s_{i+1})$$

where $\gamma$ is a *discount factor* between $(0, 1)$ (or 1 when we can guarantee certain conditions, known as a **proper policy**, this works when an MDP is set up such that all policies eventually reach a goal state). The use of discounting (recall Calculus I) makes it so even infinite chains of rewards yield a finite expected reward.

$$\sum_{i=0}^{\infty} \gamma^i R(s_i, a_i, s_{i+1}) = \frac{R_{\max}}{1-\gamma}$$

One thing that is striking about using discounting and infinite horizon, is that the starting state doesn't matter! Since all policies need to take into account all states, and there is no global counter, the action we take at any state will be the same regardless of where we started. This also mean it yields a *stationary* policy since the expected reward is symmetric between different times.

## 5 Algorithms for MDPs

### 5.1 Bellman Equation

Finally we get to the question of determining an optimal policy for an MDP. The key equation that we will use for this task is known as the Bellman equation. This equation tells us the expected future score of the optimal policy from any state in the problem.

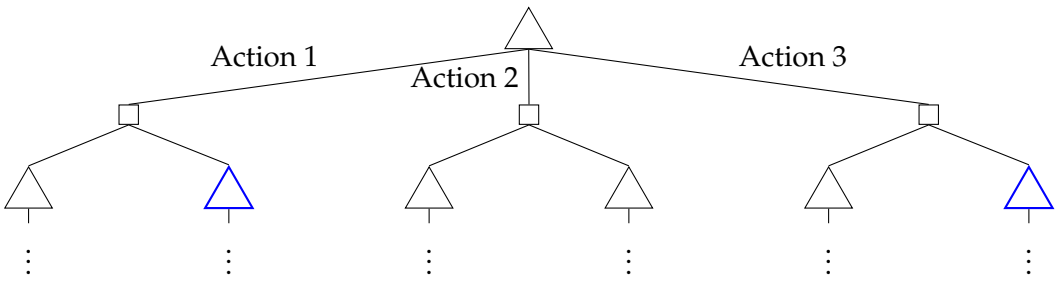



*Figure 2: Expectimax interpretation of Bellman equation.*

$$U(s) = \max_{a \in \text{ACT}(s)} E_{s'|s,a}(R(s,a,s') + \gamma U(s'))$$

Note that this equation is just a formalized version of the ExpectiMax algorithm over MDPs shown in Figure 2. We are taking a max over actions, an expectation over states, and then recursing back to the same function.

Now obviously computing this naively for infinite histories would be completely impossible. However, we can take advantage of the fact that since the policy is stationary, the value of $U$ only depends on the state. Therefore if there are multiple nodes in the search tree with the same state, we only need to calculate the corresponding $U$ once and *cache* its value in a table $U[s]$. (This same approach could help speed up game playing as well, if you are careful about the horizon). This strategy is known as *dynamic programming*.

### 5.2 Solving Darts

Now lets consider the Darts problem explicitly. There is one state for each of the possible scores $0, \ldots, 10$ and so we need to compute 11 values of $U$. Each of these can be computed by applying Bellman's equation.

Let's start by making a simplifying assumption. In standard darts, scoring 4 when there are 3 points needed will leave you at 4. If we do away with this, and say scoring 4 with 3 point needed gets you to 0, then Darts values always go down. We can then start from the goal and go compute values upwards. This will guarantee that all $U[s']$ will be already computed by the time we get to a higher (if $s < s'$ then $p(s'|s,a) = 0$ for all $a$). This is the classic case of dynamic programming.

1: **procedure** DARTS($\gamma$)
2: $U[s] = 0$ for all $s \in \mathcal{S}$
3: **for** $s = 0 \ldots 10$ **do**
4: $U[s] \leftarrow \max_{a \in \text{ACT}(s)} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma \times U[s'])$
5: **return** $U$

However this is a special case. In general there will be loops (for instance in Pacman) and we will not be able to fully precompute the expected cost for all future states. For instance this would fall apart if we go back to standard darts, since we could not easily order the states. We say this case has *loops*.

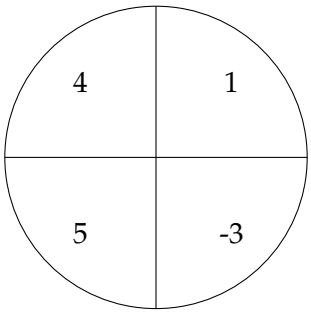
For instance to compute the value at $U[1]$, we would compute need to recursively compute:

$$U[1] = \frac{1}{2}(-1) + \frac{1}{2}(-1 + U[1])$$

The main approach we will use is iteratively repeatedly compute this value until it reaches a **fixpoint**.

```
1: procedure DARTS(γ)
2:     U[s] = 0 for all s ∈ S
3:     for s = 0 ... 10 do
4:         for t = 0 ... T − 1 do
5:             U[s] ← max_{a∈ACT(s)} Σ_{s'∈S} p(s'|s, a)(R(s, a, s') + γ × U[s'])
6:     return U
```

Here we have added an inner loop such that each value $U[s]$ can be recursively updated as it changes.

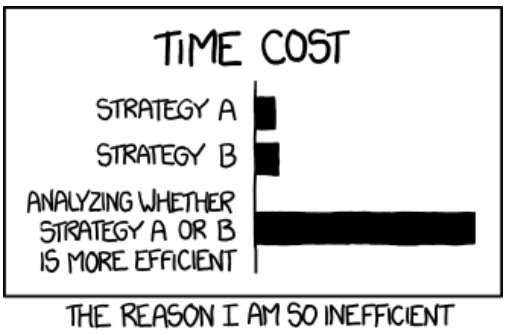



### 5.3 Value Iteration

In general we cannot do a single pass through the data to compute the correct $U$ values, however we can extend the dynamic programming idea to the case with loops. The main change will be to extend the above algorithm to run iteratively. We start with the values all equal to zero, then each time through we run dynamic programming, pretending the previous iteration's $U$ values are correct (they are not).

```
1: procedure VALUEITERATION(γ, T)
2:     U^0[s] = 0 for all s ∈ S
3:     for t = 0 ... T − 1 do
4:         for s ∈ S do
5:             U^(t+1)[s] ← max_{a∈ACT(s)} Σ_{s'∈S} p(s'|s, a)(R(s, a, s') + γ × U^(t)[s'])
       return U^T
```

This algorithm may require hundreds of iterations to run (as opposed to just one above), where each time it requires a DP pass of time $O(|\mathcal{S}|^2|\mathcal{A}|)$ in the worst case. What is somewhat surprising though is that you can show the following:




**Theorem 1** *Value iteration algorithm will eventually converge to the optimal policy.*

Proof Sketch: The proof is based on the idea of a contraction. You can show that at each iteration the maximum difference between the last values and the next values decreases.

$$\max_s |U^{(t+1)}[s] - U^{(t)}[s]| \leq \max_s |U^{(t)}[s] - U^{(t-1)}[s]|$$

Eventually it has to reach a *fixpoint*.

$$\max_s |U^{(t+1)}[s] - U^{(t)}[s]| = \max_s |U^{(t)}[s] - U^{(t-1)}[s]|$$

We can show that the only value function with a fixpoint is the true value function.

Once we have obtained the value function, the optimal policy can be computed for all $s \in \mathcal{S}$ as:

$$\pi^*[s] = \arg\max_{a \in \text{ACT}(s)} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma \times U[s'])$$

### 5.4 Policy Iteration

An alternative method to value iteration is to instead work directly with our current policy $\pi$. Given any fixed policy we can compute the value of any state by using the following equations for all $s$ in $\mathcal{S}$:

$$U^\pi(s) = \sum_{s' \in \mathcal{S}} p(s'|s,\pi(s))(R(s,a,s') + \gamma U^\pi(s'))$$

Note that there is no longer a max over actions since we are assuming the action is determined by the policy $\pi$. In this equation

- $p$ is a fixed transition model.
- $R$ is a fixed reward function.
- $\gamma$ is a fixed discount term.

The only variables are $U^\pi(s)$ of which there are $|\mathcal{S}|$. So this is a linear system of equations, with $|\mathcal{S}|$ variables, and $|\mathcal{S}|$ unknowns. This means we can solve for each of the variables using standard linear methods. We will call this operation POLICYEVAL. (On the homework, you will solve this linear system by hand).

Policy iteration works by repeatedly alternating between calling PolicyEval to calculate the values of $U^\pi$ and finding a new policy that maximizes the current $U$ function. It ends when it reaches a point when the policy is no longer changing.

---

1: **procedure** POLICYITERATION($\pi^{(1)}$)
2: **for** $t = 1 \ldots$ **do**
3: $U^{(t)} \leftarrow$ POLICYEVAL($\pi^{(t)}$)
4: **for** $s \in \mathcal{S}$ **do**
5: $\pi^{(t+1)}[s] \leftarrow \arg\max_{a \in \text{ACT}(s)} \sum_{s' \in \mathcal{S}} P(s'|s,a) U^{(t)}[s']$
6: **if** $\pi^{t+1} = \pi^t$ **then return** $\pi^{(t+1)}$

---

# 6 Reinforcement Learning

As a final application of MDPs we consider the task of reinforcement learning. Reinforcement learning is a rather deep topic with many applications and areas. It is also an topic of much current research and study. We will provide a very brief introduction to one aspect of reinforcement learning (model-free, Q-Learning), which you will experiment with on the homework. For a more in-depth coverage of the learning aspects of the topic, we highly recommend taking CS 181 or CS 283 which covers the topic at a graduate level.

## 6.1 Problem Setup

When moving from classical search to MDPs we substituted the deterministic transition model for a Markov model and substituted step costs for rewards. This made the problem much harder, but we still were given the transition distribution and the reward function itself. In reinforcement learning we further drop these two elements from the specification.

| Name | Type | Description |
|---|---|---|
| State space | $\mathcal{S}$ | The set of all possible states of the world. Still Known. |
| Action space | $\mathcal{A}$ | The set of all possible actions of the world. Still Known. |
| Action model | $\text{ACTIONS} : \mathcal{S} \mapsto 2^{\mathcal{A}}$ | Specifies which actions are applicable at a given state. |
| (Markov) Transition model | $p(s'\|s, a)$ | Not known, but observed |
| Goal state | $\text{GOAL} : \mathcal{S} \to \{0, 1\}$ | Predicate indicating if a state is terminal. |
| Reward Function | $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ | Not known, but observed |

This makes things really tough. We have a world, we can know where we are and what we can do. To actually understand the world dynamics though we have to experiments. This literally means wander around and try things out to see what rewards we can obtain. We will then use machine learning to estimate the information we need to make decisions.

### 6.2 Model-Based Learning

*Model-based* learning follows most closely to the other approaches outlined in this note. In a model-based approach, we will obtain samples about the underlying world (for instance as series of trajectories and their corresponding reward). We can then use these trajectories to estimate the transition model and the reward function.

For instance if we had enough trials to observe every transition from state $s$ to state $s'$ when taking action $a$ we would know the reward function $R$. Estimating the transition model is harder, but again with enough data you could construct a reasonable estimate.

The benefit of model-based learning is that once you have an estimate of the model then the whole problem reduces to a standard MDP. Reinforcement learning can then be solved by:

- Collecting sample from the world.
- Estimating the transition model and reward function.
- Using value iteration or policy iteration to find a policy.
- Using that policy in practice.

### 6.3 Model-Free Learning and Q-Learning

However, in practice it can be very difficult to get enough samples to construct a realistic version of a model for the purpose of value iteration. An alternative approach, *model-free* learning, uses

our trajectories to directly estimate a policy itself. If our end goal is a policy, there is no need to ever construct the MDP directly. In fact all we need is a way to assign a value to taking an action in a given state. If we estimate this we can be greedy and just use this function for future action decisions.

The function we use is called the Q-function. It is defined recursively as follows (analogously to the Bellman equation, though note that the max and expectation have been reversed and it takes a state and action as arguments):

$$Q(s,a) = \mathbb{E}_{s'|s,a} R(s,a,s') + \gamma \max_{a' \in \text{ACTS}(s')} Q(s',a')$$

We can relate the Q-function and the U-function as:

$$U(s) = \max_{a \in \text{ACTIONS}(s)} Q(s,a)$$

In value iteration, solving for the table $U$ was enough to give us a policy:

$$\pi^*[s] = \arg\max_{a \in \text{ACT}(s)} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma \times U[s'])$$

However now we no longer know $p$ or $R$ so even if we knew the value of a state it would not help. Estimating the Q-function gets around this issue.

$$\pi^*[s] = \arg\max_{a \in \text{ACT}(s)} Q[s,a]$$

Instead of estimating $U[s]$ as in value iteration, we will now estimate $Q[s,a]$.

We learn the Q-function by observing sample movements through a world. After taking each action, we compare the observed transition $(s,a) \to s'$ and reward $R(s,a,s')$ to our previous understanding of estimate of its value. We then modify the Q-function based on the difference. This is known as a *Temporal Difference* update.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s,a,s') + \gamma \max_{a' \in \text{ACTIONS}(s')} Q(s',a') - Q(s,a))$$

Where $\alpha$ is the update rate. With enough samples, this style of update will converge to a fixed estimate.

Here is how the full algorithm works. At each stage the agent calls Q-LEARNING with the last state and action $s$ and $a$ as well as the reward obtained $r$ and the new state $s'$.

---

1: **procedure** Q-LEARNING(s, a, r, s')
2: $\quad Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
3: $\quad$ **return** $\arg\max_{a' \in \text{ACTIONS}(s')} Q(s',a')$

---

That is mostly it. The agent updates the current Q-parameters and returns the highest-scoring value under the current Q-function. If this were to convert to the optimal value then the agent will just use the Q-function to play.

### 6.4 Exploration versus Exploitation

An important difference in reinforcement learning versus all the previous models we have considered is that the decisions it makes (the policy) is changed online (there is no distinction between planning and execution). For search and MDPs we constructed the optimal solution or policy before doing anything else in the problem. For reinforcement learning the agent is moving around in the world while we learn and update parameters.

This setup means that while moving around the agent only learns from about the past actions it has taken. For naive Q-learning, this means that if one action seems very promising at a state the agent will never take any of the others (when put in that state). On one hand this strategy is nice since it will continue to reap rewards from that state. On the other hand, it is conservative since it will never see any of the other (possibly better) actions.

This trade-off, known as Exploitation versus Exploration, is an issue for all reinforcement learning strategies. It analogous to the local optima problem in local search since the current strategy represents a plateau that naive q-learning can not get out of. Similarly to local search we can extend our algorithm with some randomness to encourage it to explore other actions.

In the $\epsilon$-greedy approach to exploration you randomly trade-off between a completely safe strategy (take the best action) and a fully random strategy (sample at uniform from the actions).

---

1: **procedure** $\epsilon-$GREEDY Q-LEARNING(s, a, r, s′)
2: $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
3: $a^* \leftarrow \arg\max_{a' \in \text{ACTIONS}(s')} Q(s',a')$
4: $p \leftarrow$ random between 0 and 1
5: **if** $p > \epsilon$ **then**
6: **return** $a^*$
7: **else return**
8: Sample from ACTIONS$(s')$

---

Implementing a sensible exploration strategy is critical for having the algorithm perform well in practice.